Head Lazy but Shapely, and Deeply Mutable

Ben Lippmeier
University of New South Wales
SAPLING 2010/11/18

Type classes are awesome!

- Read, Show, Eq, Ord, Functor, Foldable, Traversable, Data, Monad, Typable, Integral, Fractional, Real, Floating .....
- All work on types of kind * or (* -> *)
- All concern *value*.

Type classes are awesome!

- Read, Show, Eq, Ord, Functor, Foldable, Traversable, Data, Monad, Typable, Integral, Fractional, Real, Floating .....
- All work on types of kind * or (* -> *)
- All concern *value*.
- Disciple has region, effect and closure types as well.
- What type classes work on *them*?

Regions, Effects and Closures

```
add :: Int -> Int -> Int
```

Regions, Effects and Closures

```
add :: Int r1 -> Int r2 -> Int r3
```

Regions, Effects and Closures

```
add :: forall r1 r2 r3
    .  Int r1 -> Int r2 -> Int r3
```

Regions, Effects and Closures

```
add :: forall r1 r2 r3
     .  Int r1 -> Int r2 -(e1)> Int r3
     :- e1 = Read r1 + Read r2
```

Regions, Effects and Closures

```
add :: forall r1 r2 r3
    .  Int r1 -> Int r2 -(e1 c1)> Int r3
    :- e1 = Read r1 + Read r2
    ,  c1 = Clo (Int r1)
```

Mutability, Constancy and Purity

```
counter :: Mutable r1 => Int r1

pi      :: Const r1   => Float r1
```

Mutability, Constancy and Purity

```
counter :: Mutable r1 => Int r1

pi      :: Const r1   => Float r1
```

```
instance Const r1 => Pure (Read r1)
```

Suspension

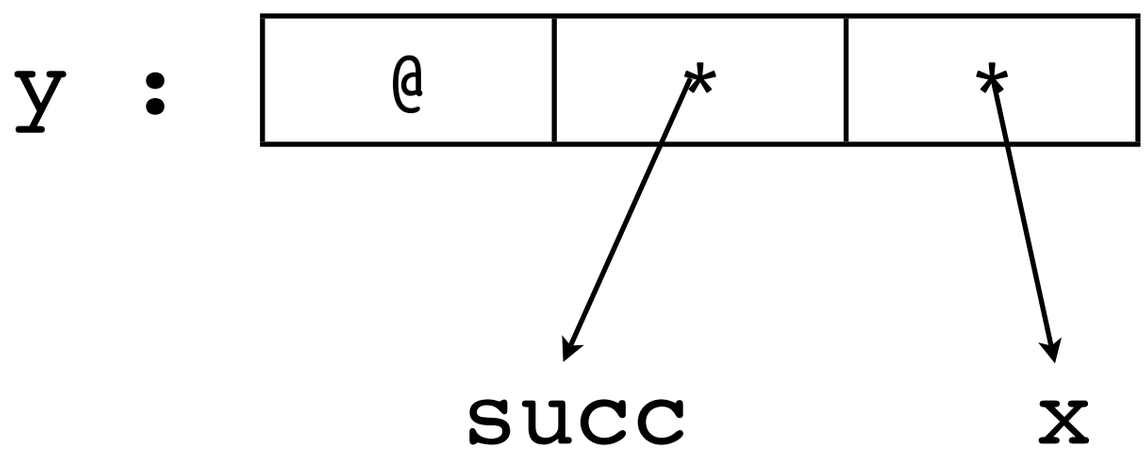
```
suspend :: forall a b e1
         .  Pure e1
        => (a -(e1)> b) -> a -> b
```

Suspension

```
suspend :: forall a b e1
        .  Pure e1
        => (a -(e1)> b) -> a -> b

y :: Int r1
y =  suspend succ x
```

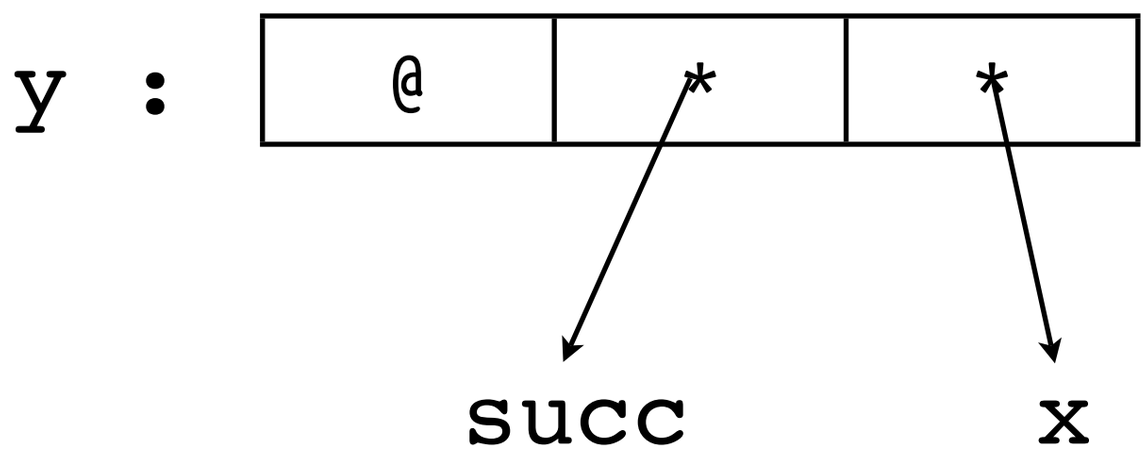
Suspension

```
suspend :: forall a b e1
        .  Pure e1
        => (a -(e1)> b) -> a -> b

y :: Int r1
y =  suspend succ x
```



Suspension

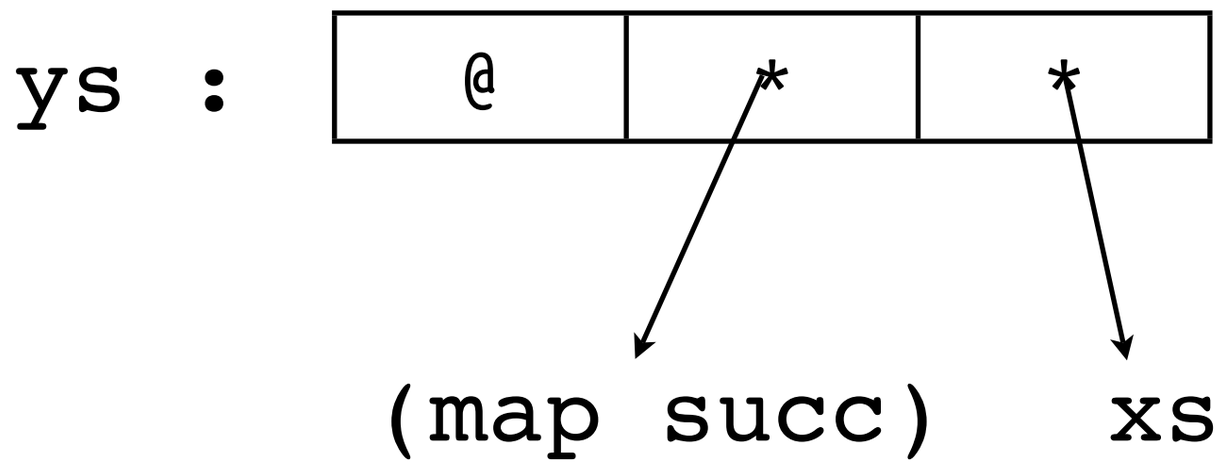
```
suspend :: forall a b e1
         .  Pure e1
        => (a -(e1)> b) -> a -> b

y :: Lazy r1 => Int r1
y =  suspend succ x
```



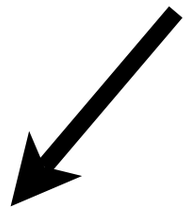
Suspension

```
suspend :: forall a b e1
         .  Pure e1
        => (a -(e1)> b) -> a -> b
```

```
ys :: Lazy r1 => List r1 (Int r2)
ys =  suspend (map succ) xs
```

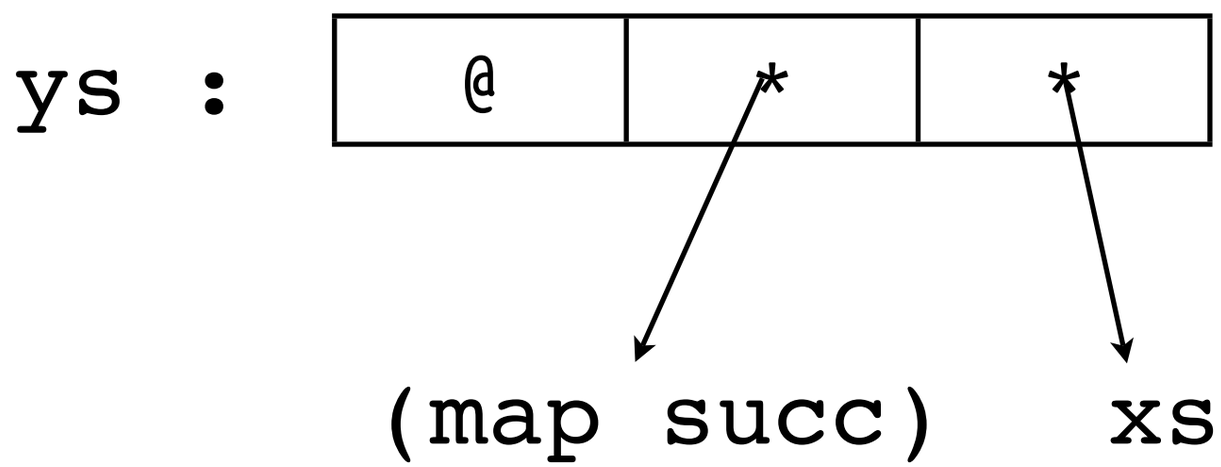


Suspension



```
suspend :: forall a b e1
         .  Pure e1, HeadLazy b
        => (a -(e1)> b) -> a -> b
```

```
ys :: Lazy r1 => List r1 (Int r2)
ys =  suspend (map succ) xs
```



Suspension

```
suspend :: forall a b e1
         .  Pure e1, HeadLazy b
        => (a -(e1)> b) -> a -> b
```

```
instance
 HeadLazy (Int r1)                    => Lazy r1
 HeadLazy (Float r1)                  => Lazy r1
 HeadLazy (List r1 a)                 => Lazy r1
 HeadLazy (Maybe r1 (Int r2)) => Lazy r1
```

Update a single Int

```
updateInt :: Int -> Int -> ()
```

Update a single Int

```
updateInt :: forall r1 r2. Mutable r1
          => Int r1 -> Int r2 -(e1)> ()
          :- e1 = Write r1 + Read r2
```

Update all the elements in a list

```
updateListInt
 :: forall r1 r2 r2 r4. Mutable r2
 => List r1 (Int r2) -> List r3 (Int r4) -(e1)> ()
 :- e1 = Read  r1 + Read  r2 + Read r3 + Read r4
       + Write r1 + Write r2

updateListInt [] _  = ()
updateListInt _  [] = ()
updateListInt (x:xs) (y:ys)
 = do updateInt x y
      updateListInt xs ys
```

The Update class

```
class Update a where
 (:=) :: forall a. DeepMutable a
      => a -> a -(e1 c1)> ()
      :- e1 = DeepRead a + DeepWrite a
      ,  c1 = Clo a
```

The Update class

```
class Update a where
 (:=) :: forall a. DeepMutable a
      => a -> a -(e1 c1)> ()
      :- e1 = DeepRead a + DeepWrite a
      ,  c1 = Clo a

instance
  DeepMutable (Int r1) => Mutable r1

  DeepMutable (List r1 (Int r2)) => Mutable r1
                                ...  => Mutable r2
```

Copying data

```
copyInt
   :: forall r1 r2
   .  Int r1 -(e1)> Int r2
   :- e1 = Read r1
```

```
copyListInt
   :: forall r1 r2 r3 r4
   .  List r1 (Int r2) -(e1)> List r3 (Int r4)
   :- e1 = Read r1 + Read r2
```

The Copy class

```
class Copy a where
  copy :: a -(e1)> a
       :- e1 = DeepRead a
```

The Copy class

```
class Copy a where
  copy :: a -(e1)> a
       :- e1 = DeepRead a
```

```
copyInt
  :: forall r1 r2
  .  Int r1 -(e1)> Int r2
  :- e1 = Read r1
```

The Copy class

```
class Copy a where
  copy :: forall b. Shape a b
       => a -(e1)> b
       :- e1 = DeepRead a
```

```
copyInt
  :: forall r1 r2
  .  Int r1 -(e1)> Int r2
  :- e1 = Read r1
```

The Copy class

```
class Copy a where
  copy :: forall b. Shape a b
       => a -(e1)> b
       :- e1 = DeepRead a
```

```
instance Shape (Int r1) (Int r2)

instance Shape a b
      => Shape (List r1 a) (List r2 b)
```

The Copy class

```
class Copy a where
  copy :: forall b. Shape a b
       => a -(e1)> b
       :- e1 = DeepRead a
```

```
instance Copy (Int r1) where
  copy x = x
```

Computing the instance type

```
class Copy a where
  copy :: forall b
       .  Shape a b
       => a -(e1)> b
       :- e1 = DeepRead a
```

Computing the instance type

```
forall a.
  class Copy a where
    copy :: forall b
          .  Shape a b
         => a -(e1)> b
         :- e1 = DeepRead a
```

Computing the instance type

```
forall (Int r1).
  class Copy (Int r1) where
    copy :: forall b
          .  Shape (Int r1) b
         => Int r1 -(e1)> b
         :- e1 = DeepRead (Int r1)
```

Computing the instance type

```
forall (Int r1).
  class Copy (Int r1) where
    copy :: forall (Int r2)
          .  Shape (Int r1) (Int r2)
         => Int r1 -(e1)> Int r2
         :- e1 = DeepRead (Int r1)
```

Computing the instance type

```
forall r1.
  class Copy (Int r1) where
    copy :: forall r2
          .  Shape (Int r1) (Int r2)
         => Int r1 -(e1)> Int r2
         :- e1 = DeepRead (Int r1)
```

Computing the instance type

```
forall r1.
  class Copy (Int r1) where
    copy :: forall r2
          .  Shape (Int r1) (Int r2)
         => Int r1 -(e1)> Int r2
         :- e1 = DeepRead (Int r1)
```

Computing the instance type

```
forall r1.
  class Copy (Int r1) where
    copy :: forall r2
          .  Int r1 -(e1)> Int r2
          :- e1 = DeepRead (Int r1)
```

Computing the instance type

```
forall r1.
  class Copy (Int r1) where
    copy :: forall r2
          .  Int r1 -(e1)> Int r2
          :- e1 = Read r1
```

Computing the instance type

```
forall r1.
  class Copy (Int r1) where
    copy :: forall r2
          .  Int r1 -(e1)> Int r2
          :- e1 = Read r1
```

```
copy :: forall r1 r2
      .  Int r1 -(e1)> Int r2
      :- e1 = Read r1
```

Computing the instance type

```
forall r1.
  class Copy (Int r1) where
    copy :: forall r2
         .  Int r1 -(e1)> Int r2
         :- e1 = Read r1
```

```
copy :: forall r1 r2
     .  Int r1 -(e1)> Int r2
     :- e1 = Read r1


instance Copy (Int r1) where
    copy x = x
```

Data types containing functions

```
data IntFun r1..r4 e1 c1
     = SInt (Int r2)
     | SFun (Int r3 -(e1 c1)> Int r4)
```

Data types containing functions

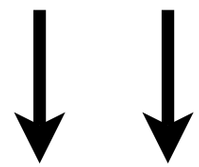
```
data IntFun r1..r4 e1 c1
     = SInt (Int r2)
     | SFun (Int r3 -(e1 c1)> Int r4)

copy[IntFun]
 :: forall r1..r8 e1 c1
    IntFun r1..r4 e1 c1
                  -(e2)> IntFun r5 r6 r7 r8 e1 c1
:- e2 = Read r1..r4
```

Data types containing functions

```
data IntFun r1..r4 e1 c1
     = SInt (Int r2)
     | SFun (Int r3 -(e1 c1)> Int r4)

copy[IntFun]
 :: forall r1..r8 e1 c1
    .  IntFun r1..r4 e1 c1
                  -(e2)> IntFun r5 r6 r7 r8 e1 c1
 :- e2 = Read r1..r4

copy xx
 = case xx of
     SInt i -> SInt (copyInt i)
     SFun f -> ???
```

Data types containing functions

```
data IntFun r1..r4 e1 c1
     = SInt (Int r2)
     | SFun (Int r3 -(e1 c1)> Int r4)

copy[IntFun]
 :: forall r1..r8 e1 c1
 .  IntFun r1..r4 e1 c1
                  -(e2)> IntFun r5 r6 r3 r4 e1 c1
 :- e2 = Read r1..r4

copy xx
 = case xx of
     SInt i -> SInt (copyInt i)
     SFun f -> SFun f
```



Material region variables

```
x :: Int rl
```

Material region variables

```
x :: forall r1. Int r1
```

✕ no!

Material region variables

```
x :: Int rl
```

Material region variables

```
    x :: Int r1

succ :: forall r2 r3
      .  Int r2 -> Int r3
```

Material region variables

```
    x :: Int r1

succ :: forall r2 r3
      .  Int r2 -> Int r3

samex :: () -> Int r1
samex () = x
```

Material region variables

```
    x :: Int r1

succ :: forall r2 r3
      .  Int r2 -> Int r3

samex :: () -(c1)> Int r1
      :- c1 = Clo (Int r1)
samex () = x
```

Material region variables

```
data IntFun r1 r2 r3 r4 e1 c1
     = SInt (Int r2)
     | SFun (Int r3 -(e1 c1)> Int r4)


   r1, r2                : material
   r3, r4, e1, c1        : immaterial


instance
 Shape (IntFun r1 r2 r3 r4 e1 c1)
       (IntFun r5 r6 r3 r4 e1 c1)
```

Material region variables

```
data IntFun r1 r2 r3 r4 e1 c1
     = SInt (Int r2)
     | SFun (Int r3 -(e1 c1)> Int r4)
```

```
r1, r2              : material
r3, r4, e1, c1      : immaterial
```

```
instance
 Shape (IntFun r1 r2 r3 r4 e1 c1)
       (IntFun r5 r6 r3 r4 e1 c1)
```

immaterial == cannot copy

Emptiness

```
spawn :: forall a b c
       .  ( Serialisable a, Serialisable b
          , Pure e1, Empty c1)
      => (a -(e1 c1) b) -> a -> b
```

Disjoint effects

```
fork
  :: forall a b e1 e2. Disjoint e1 e2
  => (a -(e1)> b) -> (a -(e2)> b) -> a -(e3)> b
  :- e3 = e1 + e2

instance
  Pure e1          => Disjoint e1 e2
  Pure e2          => Disjoint e1 e2

  Distinct r1 r2 => Disjoint (Read  r1) (Read  r2)
  Distinct r1 r2 => Disjoint (Read  r1) (Write r2)
  Distinct r1 r2 => Disjoint (Write r1) (Read  r2)
  Distinct r1 r2 => Disjoint (Write r1) (Write r2)
```

Questions?